

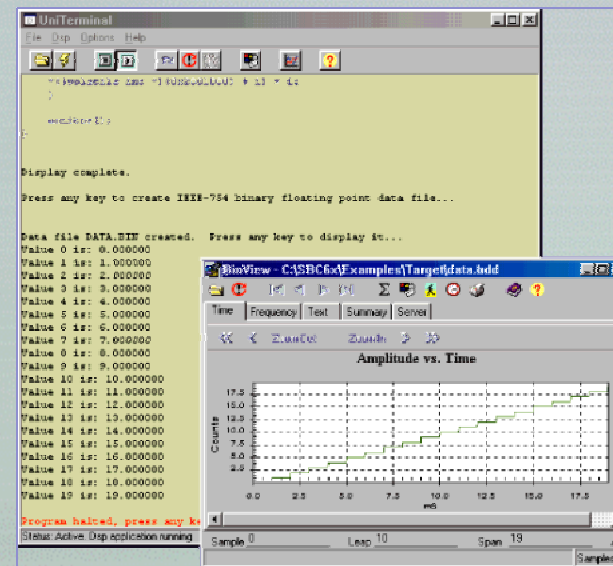
# Zuma Training/Tutorial





# Innovative Integration's Zuma Toolset

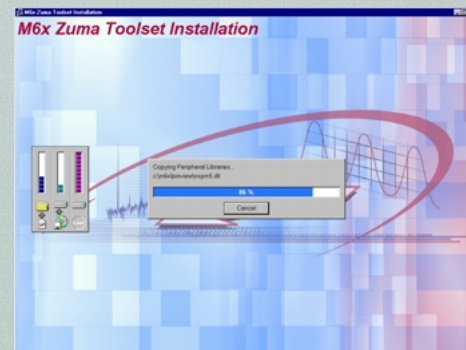
Comprehensive tools and libraries for  
DSP applications





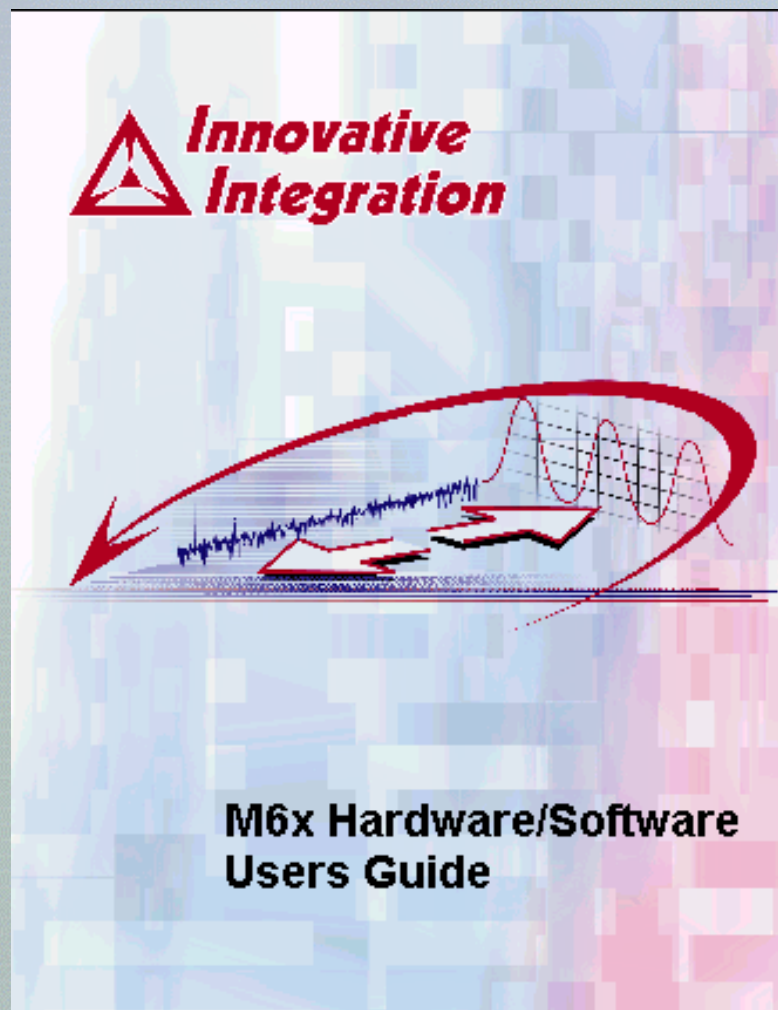
# Getting Started

- Necessary components
  - CCStudio compile tools are needed to compile programs for all TI DSP cards
  - JTAG emulator and CCStudio debug tools are necessary for debugging code on TI DSPs
    - Innovative Integration's Code Hammer JTAG emulator or XDS510 compatible
- Zuma is bundled with CCStudio
  - Reduces installation errors
  - Ensures the proper version is used
  - Simplifies installation – only one CD needed





# Getting Started

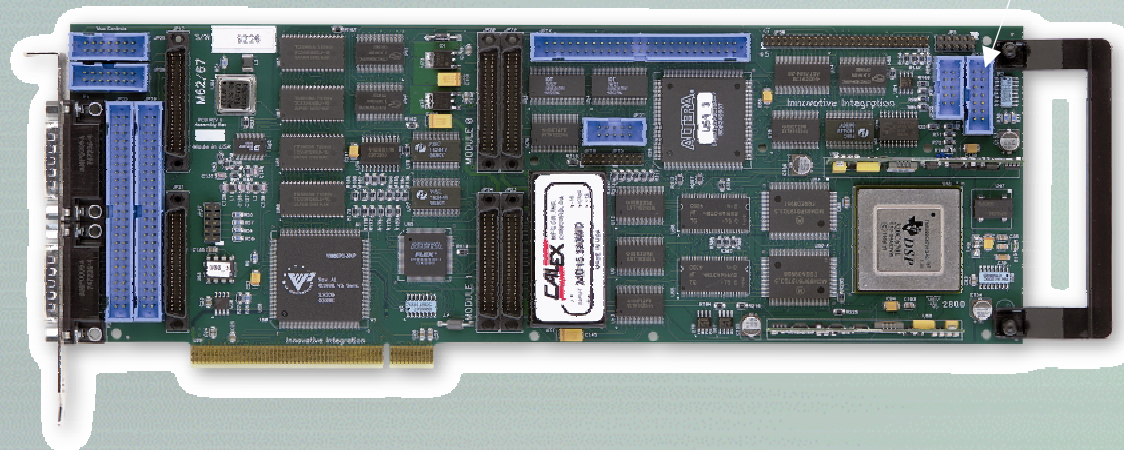


- Insert the II CD and run the install for the M6x under the PCI/cPCI products tab.
  - Follow the installation instructions in the manual.
  - You should select the PCI JTAG
  - There is no need to install the Borland or Visual Basic components.
  - Skip the JTAG Debugger Driver Installation section.
  - The typical installation will also install CCStudio and you will need to enter a password.
  - At the end of the CCStudio installation you must chose **not** to restart your machine at this time. The installation will continue after this.



# Getting Started

- Now turn off the machine and install the JTAG card and the M6x board.
  - The POD should be connected to the JTAG port on the card and the A4D4 should be installed in site 0 before you power up.

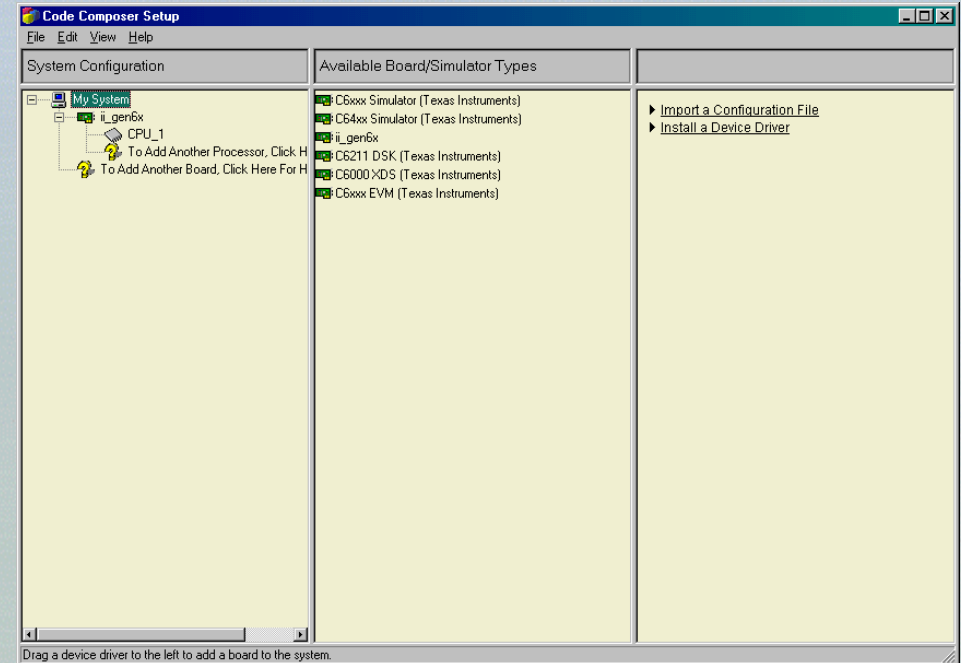


JTAG Port



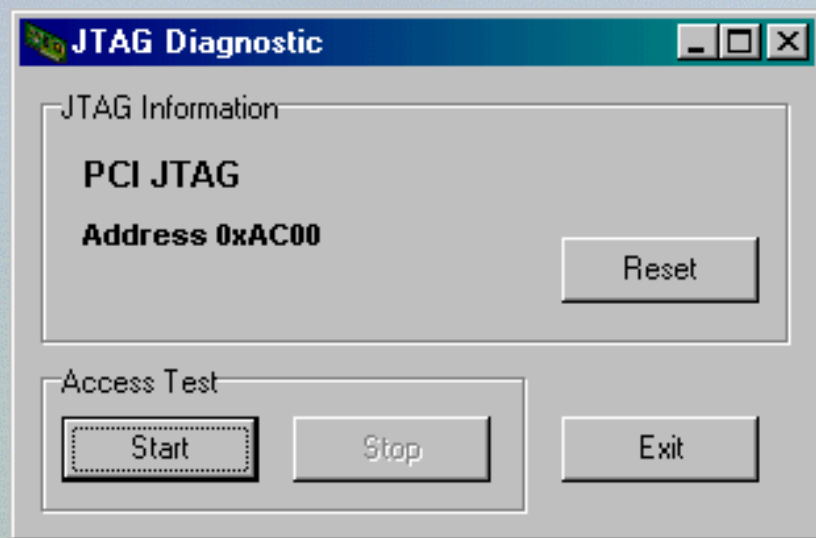
# Getting Started

- When you power up, the CCStudio setup program may start automatically, close it (exit) and the II installation will set it up automatically or it may be done manually later.
  - It is best to set up later since you will not know the address of the JTAG until the board is recognized by windows.





# Getting Started

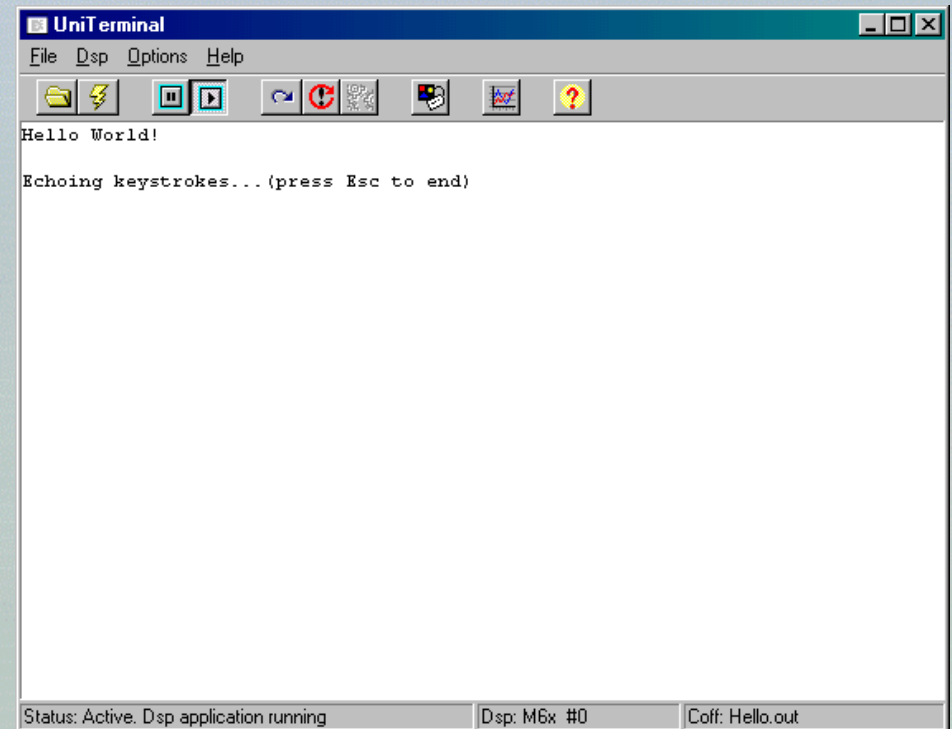


- Test the JTAG Debugger installation by running JTAGDIAG from the start menu under M6x DSP board.
  - Pressing the Start button will blink the ACCESS LED on the JTAG POD if the driver is connected properly.



# Getting Started

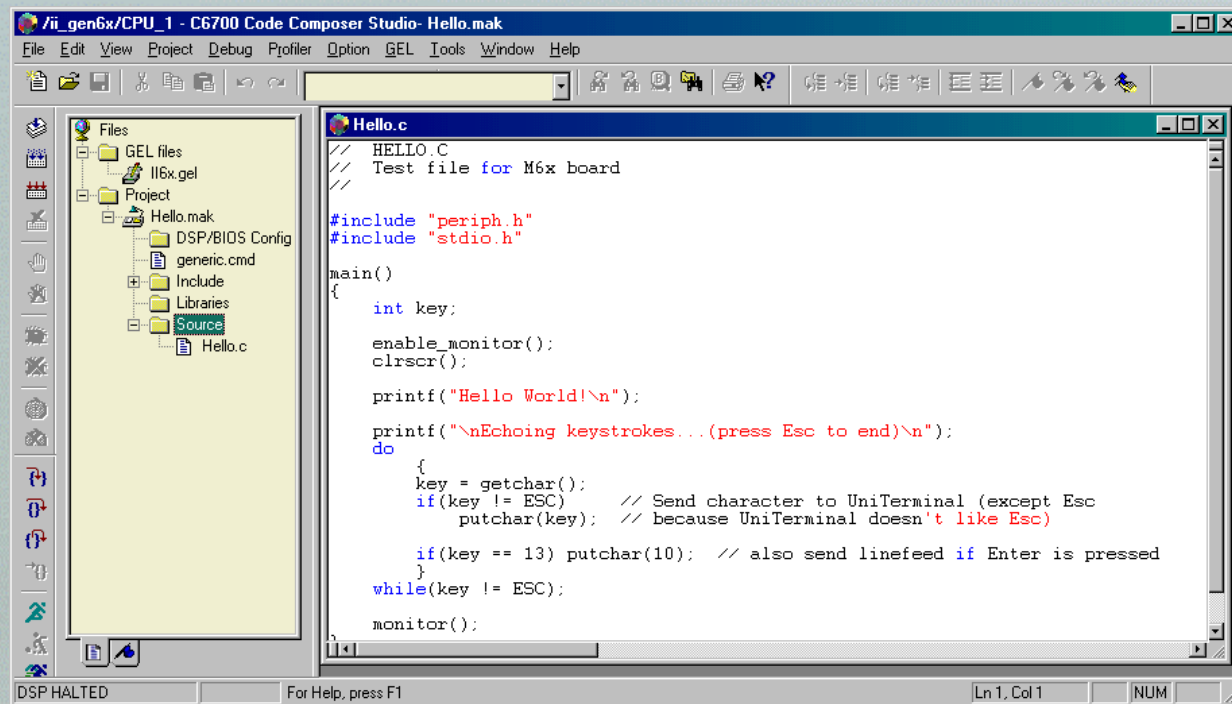
- Next Open UniTerminal from the Windows start menu.
  - From the File menu do a Coff file ->Download and load the Hello.out file in:  
M6x/examples/target
  - The screen should look the same as shown on the right.





# Getting Started

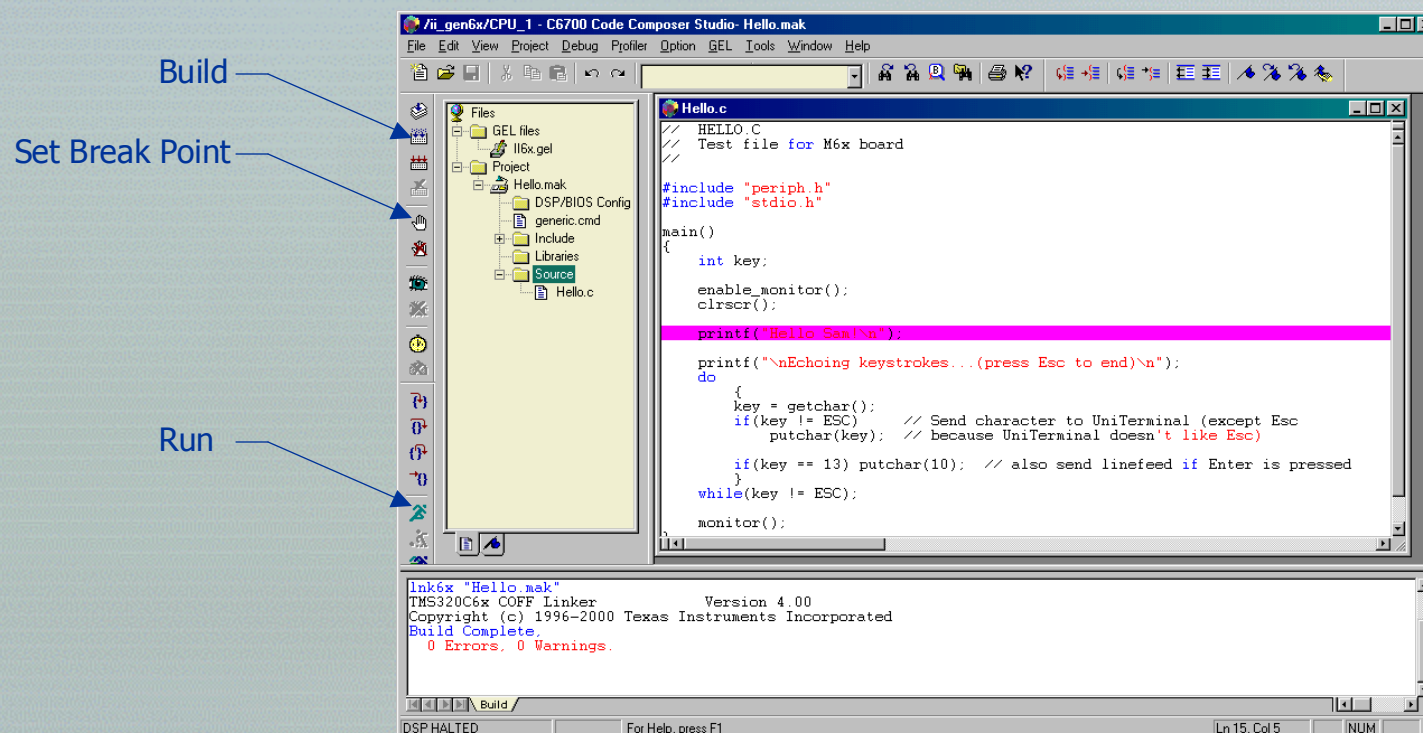
- Leaving UniTerminal running, Open CCStudio from the windows start menu.
- From the Project menu, select Open and open Hello.mak in the M6x\examples\target directory.
- Open the source code by expanding the Project folder on the left, expanding hello.mak project, expanding the source code folder, then double clicking the hello.out file.





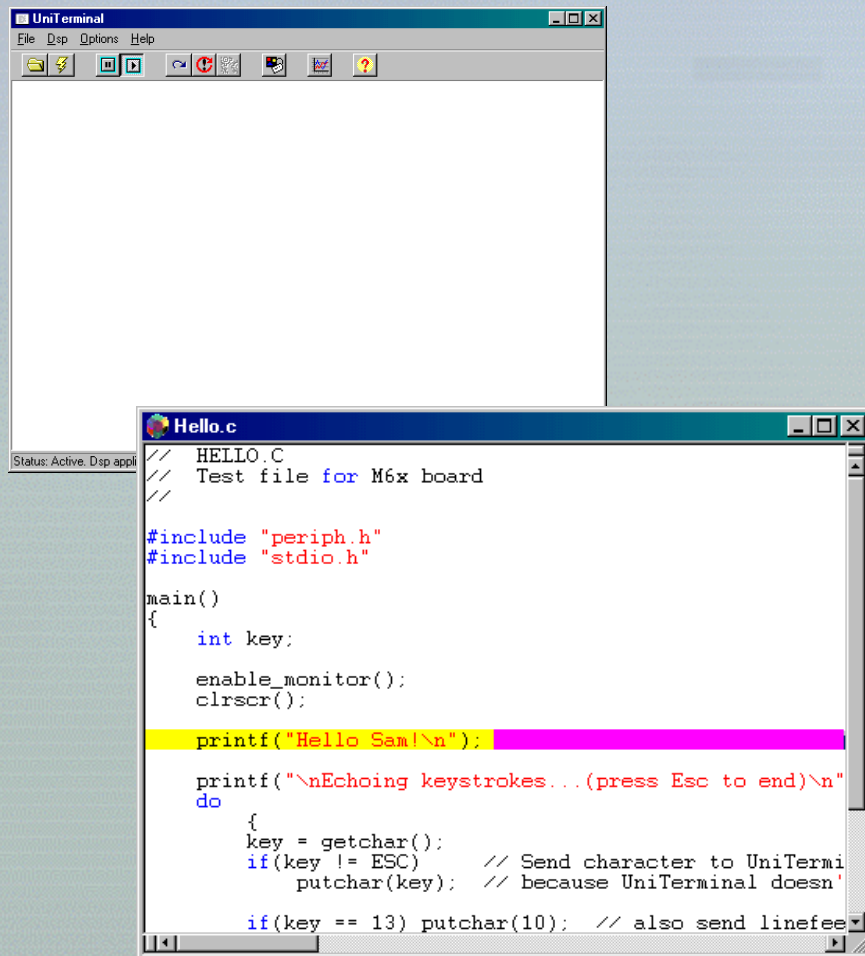
# Getting Started

- In the source code window, change "Hello World" to "Hello <your name>".
- From the Project menu select Build.
  - You should get notification soon that your application rebuilt with no errors.
- Place the cursor on the Hello line and left click to move the cursor to this line. Click the white hand on the left to set a break point.





# Getting Started



The image shows two windows from a software application. The top window is titled 'UniTerminal' and has a menu bar with 'File', 'Dsp', 'Options', and 'Help'. Below the menu is a toolbar with icons for file operations. The bottom window is titled 'Hello.c' and contains C code. The code includes comments and standard headers, and defines a main function that prints 'Hello Sam!' and enters a loop to echo keystrokes. The line `printf("Hello Sam!\n");` is highlighted with a yellow background, and the line `printf("\nEchoing keystrokes...(press Esc to end)\n"` is highlighted with a red background.

```
// HELLO.C
// Test file for M6x board
//
#include "periph.h"
#include "stdio.h"

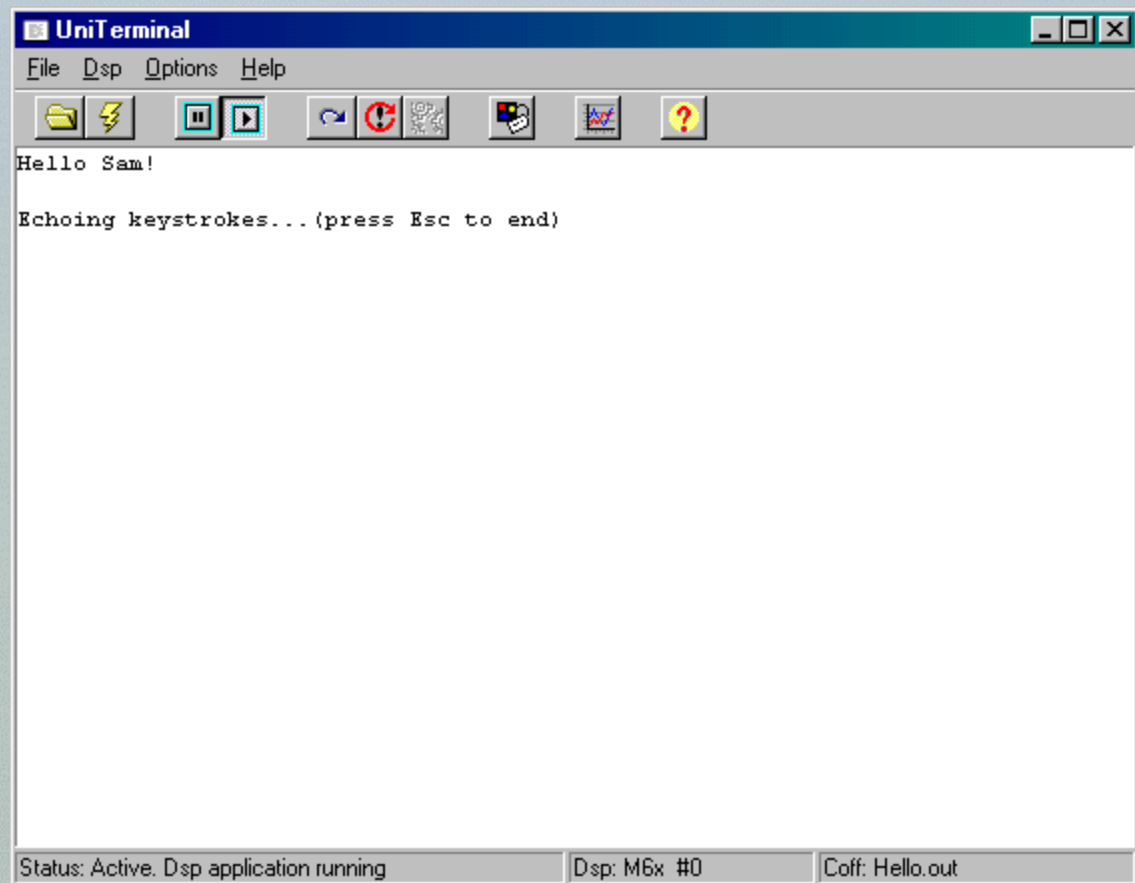
main()
{
    int key;
    enable_monitor();
    clrscr();
    printf("Hello Sam!\n");
    printf("\nEchoing keystrokes...(press Esc to end)\n"
    do
    {
        key = getchar();
        if(key != ESC)    // Send character to UniTermi
            putchar(key); // because UniTerminal doesn't
        if(key == 13) putchar(10); // also send linefeed
    } while(key != ESC);
}
```

- From the File menu, select File Download. Browse to the Hello.out file that was just rebuilt and click Open.
- After the file downloads, press the Green Man icon on the left and the program will run to the Hello line and stop.
  - At this point the UniTerminal screen should be clear and the hello line should be half yellow and half red indicating that the processor has stopped on that line.



# Getting Started

- Press the Green Man icon again and the program will continue to run and the UniTerminal will show the appropriate output.





# Zuma Toolset



- Technical Summary
- Usage Examples
- DLL Overview
- DspComponent

**ZUMA**  
TOOLSET





# Zuma Toolset Technical Summary



# Terminology

- Host
  - PC running Windows equipped with software applications specifically designed to allow development of application programs for DSP-equipped targets.
- Target
  - Small, self-contained, “microcomputer-on-a-card”
  - Features a digital signal processor (DSP) able to perform control and data acquisition functions
  - Three basic types:
    - SBC: Able to run without a PC
    - PCI: Requires a PCI slot within a conventional PC
    - cPCI: Requires a PCI slot within a CompactPCI PC



# Terminology

- Native Development
  - Use Host SW to build applications that run on the Host
    - BCB, MSVC, MSVB, Delphi, et al.
- Cross Development
  - Use Host SW to build applications that run on the Target
    - Code Composer, TI Compiler, TI Assembler, et al
- PCI and cPCI Targets
  - Usually requires both Native and Cross development
- SBC Targets
  - Usually requires just Cross development



# Native Development



- Compiler
  - Converts source (C/C++/Pascal) language source files (.c/.cpp/.pas) into assembly language files (.asm)
- Assembler
  - Converts assembly language files (.asm) into machine language files (.obj).
- Linker
  - Combines machine language files (.obj/.dcu) with library files (.lib/.dll) to create target executable files (.exe)



# Native Development



- IDE
  - Integrated set of tools to support many aspects of application software development
    - Editor
      - Authoring of application source code
    - Compiler/Linker/Assembler
      - Conversion of source into executable code
    - Debugger
      - Seizes control of Host CPU to permit rapid discovery and correction of software defects in executable code



# Native Development



- Zuma DLL (Dynamic Link Library)
  - Library of Host functions used to allow user-written application programs to interact with DSP target boards
  - Usable from within virtually any language
    - C/C++, Visual Basic, Delphi, etc
- Note
  - DOES NOT provide a means of using the DSP board for any particular purpose "out-of-the-box"
  - DOES provide a vehicle for basic target access, initialization and control



# Native Development



- DspComponent
  - Simplified interface to any Zuma Host DLL
    - Faster and easier development
    - Less Host and Target programming knowledge needed
  - Packaged as a drag-n-drop “component”
    - VCL for Borland Builder
    - ActiveX for MSVC, MSVB and others
  - Facilitates target-independent applications, like UniTerminal



# Native Development



- Process
  - Write/modify source code using editor
    - Combine:
      - Windows API, Zuma DLL or DspComponent and custom functions
  - Convert source into executable (.exe) using compiler
  - Test executable under debugger
  - Iterate 1..3 until defects eliminated



# Cross Development



- Tools
  - C/C++ Compiler
    - Converts C/C++ language source files (.c) into assembly language files (.asm)
  - Assembler
    - Converts assembly language files (.asm) into machine language files (.obj).
  - Linker
    - Combines machine language files (.obj) with library files (.lib) to create target executable files (.out)



# Cross Development

- Tools (cont)
  - Debugger
    - Seizes control of target DSP to permit rapid discovery and correction of software defects
  - JTAG Debugger
    - Type of debugger which controls the target using a dedicated hardware communications channel, JTAG 1149.1
  - Applet
    - Utility program running on the Host PC to permit a specific development activity



# Cross Development



- Tools (cont)
  - Target-Independent
    - Standard I/O
      - Popular C library functions allowing target programs to print characters to the screen and read characters from the keyboard (like DOS applications)
    - UniTerminal
      - Applet which steers target standard I/O to the Host keyboard and screen
    - BinView
      - Applet which permits graphing of binary data stored in Host files or memory
    - Download
      - Applet which configures a PCI or cPCI-type target to run a user-written target application at Windows startup



# Cross Development

- Tools (cont)
  - Target-Independent
    - JtagDiag
      - Applet which initializes Innovative JTAG interface board.
    - CoffDump
      - Applet which reports memory usage of any target application



# Cross Development

- Tools (cont)
  - SBC-Specific Tools
    - Burn
      - Applet which configures an SBC-type target to run a user-written application at power-on
    - PromImage
      - Applet which converts a user-written target file (.out) into format needed by Burn applet (.bin)
    - ComConfig
      - Applet which configures the default communications speed (baud rate) between SBC-type targets and Host applets



# Cross Development

- Tools (cont)
  - M6x-Specific Tools
    - Boot
      - Applet which downloads and runs benign program on target.  
(Debugger aid).
  - Q6x-Specific Tools
    - QBoot
      - Applet which downloads and runs benign program on target.  
(Debugger aid).
    - UniTerminal
      - Supports .MPO files (clusters of up to four .OUT files) downloaded as  
a unit



# Cross Development



- Zuma Peripheral Libraries
  - Library of Target functions used to allow user-written application programs to interact with DSP peripherals:
    - A/Ds, D/As, Digital I/O, UARTs, PCI bus, etc.
- Note
  - DOES NOT provide a means of using the DSP board for any particular purpose "out-of-the-box"
  - DOES provide a basis for sophisticated target peripheral access, initialization and control

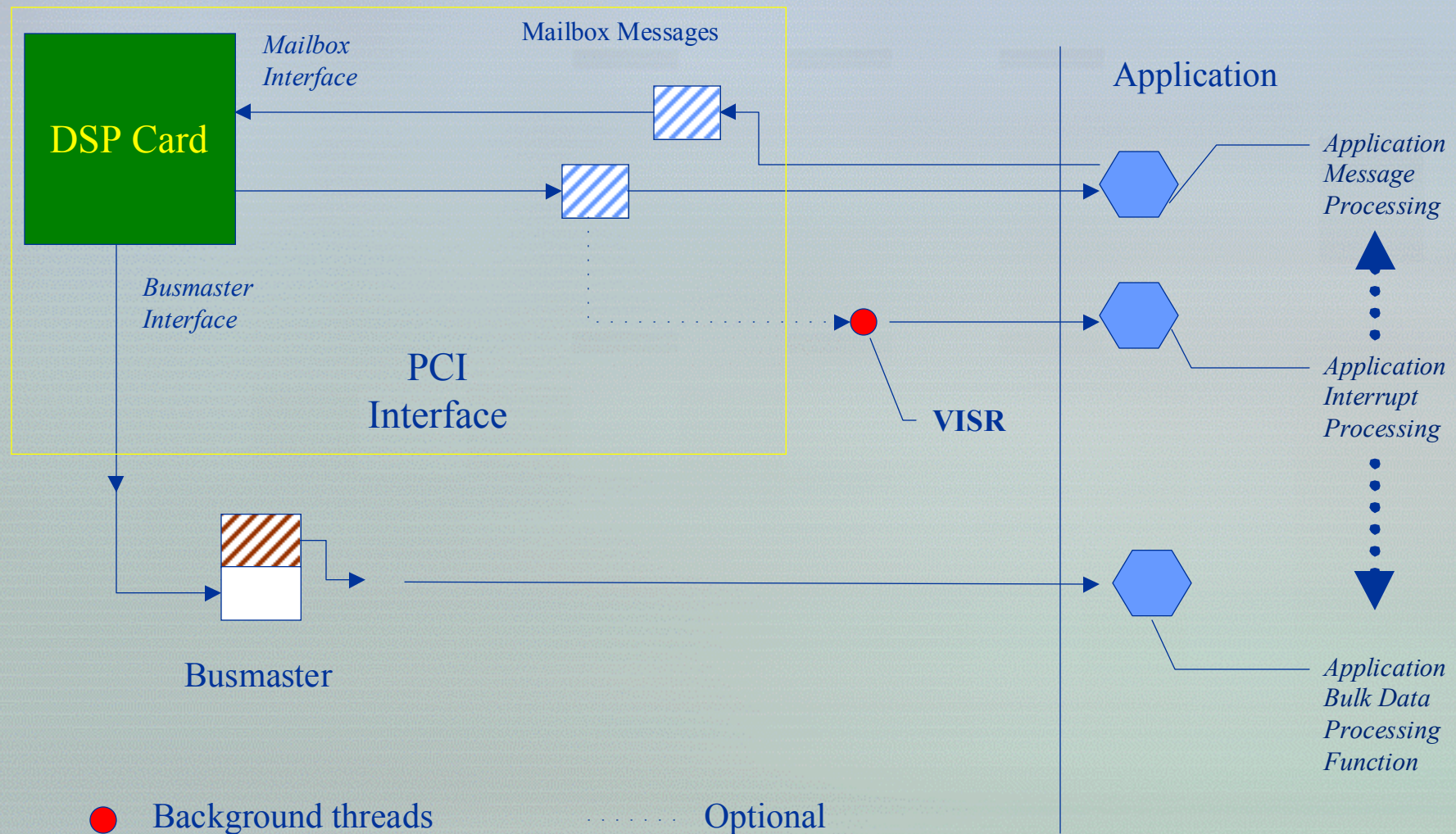


# Cross Development

- Process
  - Write/modify source code using editor
    - Combine:
      - TI C, Zuma Peripheral or custom functions
  - Convert source into executable (.out) using compiler
  - Test executable under debugger (Code Composer)
  - Iterate 1..3 until defects eliminated
- PCI/cPCI: Download executable via DLL from within Host application or at Windows startup
- SBC: Burn executable in ROM. Boot from ROM at power-on

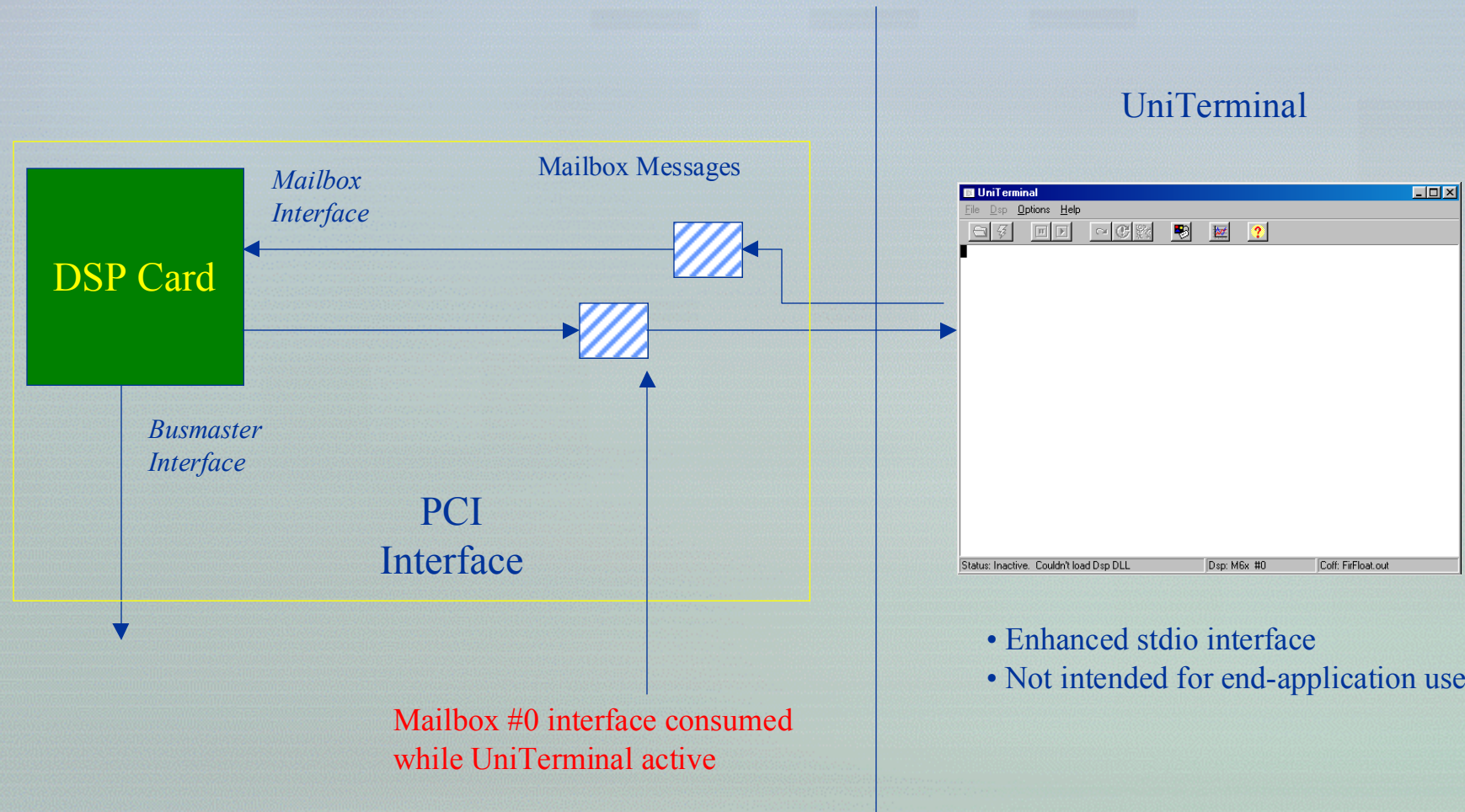


# Zuma Application Model



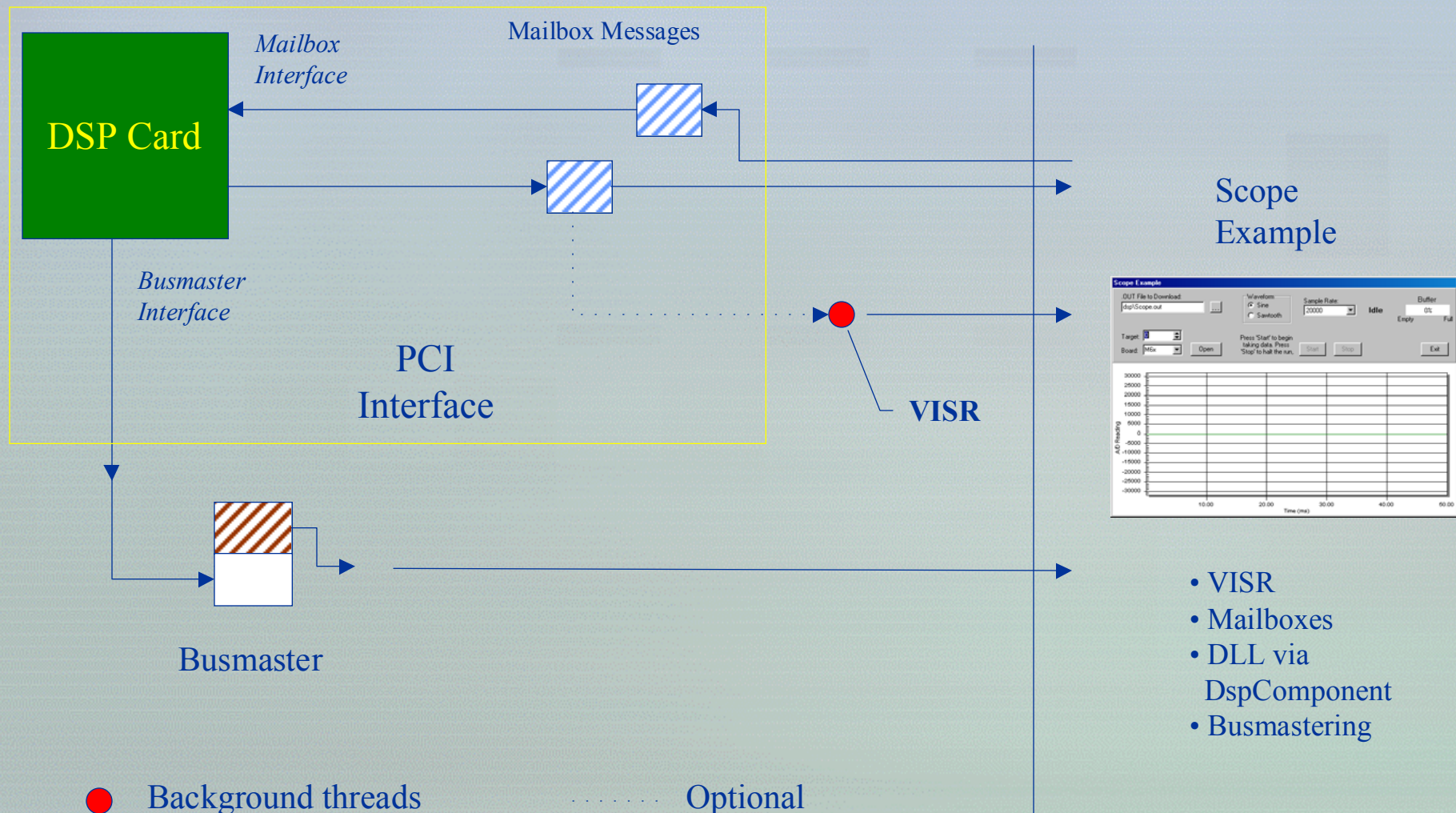


# UniTerminal Model





# Application Model



- VISR
- Mailboxes
- DLL via  
DspComponent
- Busmastering



# Example Target Application



- Hello World
  - Start UniTerminal. Target Init OK?
  - Start CCStudio
  - Project | Load \M6x\Examples\Target\Hello.mak
  - Click Project | Rebuild All
  - Click File | Load Program to download the executable
  - Click Debug | Run Free to run the executable
  - Observe output on Terminal display.




# Modify Target Application

- The world is a harsh mistress. Edit the target source (hello.c) to display:  
  
    " Goodbye, cruel World!"
- Recompile, download and run the modified code



# Example Host Application

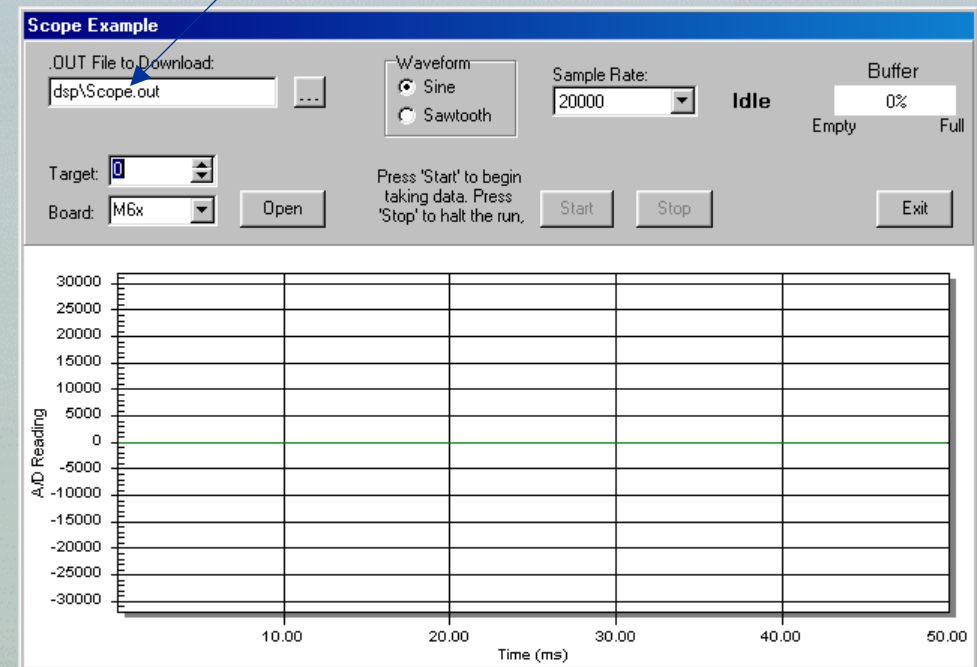
- Scope
  - Start BCB
  - Load \M6x\Examples\Host\Scope\Scope.bpr
  - Click File | Build Scope
  - Click  to run the example



# Example Host Application

- Scope
  - Similar complexity to Armada ArbGen example
  - GUI-specific code in ScopeMain.cpp
  - Target-Specific code in DspBoardFtns.cpp (326 lines of code)

Verify COFF location





# Class Project

- Introductory Example Sequence
  - Start Host IDE
  - Create new Project
    - Select File | New Application from the Menu
    - Select File | Save All
      - Name Project "ZumaDemo"
      - Name Unit1 "Main"



# Class Project



- The Goal
  - Communicate with a target application
    - Demo1.c
- Protocol
  - Host Synchronization
    - Keeps from losing commands
  - Command Loop
    - Check for Mailbox Data
    - Read Command Word
      - Reset Counter Command
    - Send Counter to Host

```
/*
 * Demo1.c      Target / Host Communication Demo
 *
 */

#include "periph.h"

const int MailboxID = 0; /* Use mailbox 0 for communication */

const int ResetCmd = 0xFFFF;

main()
{
    int data = 0; /* Used to save mailbox data read from host */
    int reps = 0; /* Counter for final loop */

    /* Turn on interrupts globally */
    enable_interrupts();

    /* Synchronize with host */
    write_mailbox(777, MailboxID);
    data = read_mailbox(MailboxID);

    /* Loop forever doing commands */
    while (1==1)
    {
        /* Check to see if the host sent command */
        if (check_inbox(MailboxID))
        {
            /* Host sent a command */
            data = read_mailbox(MailboxID);

            /* If a reset command was sent, set reps to 0 */
            /* otherwise, increment reps */
            if (data == ResetCmd)
                reps = 0;
            else
                ++reps;

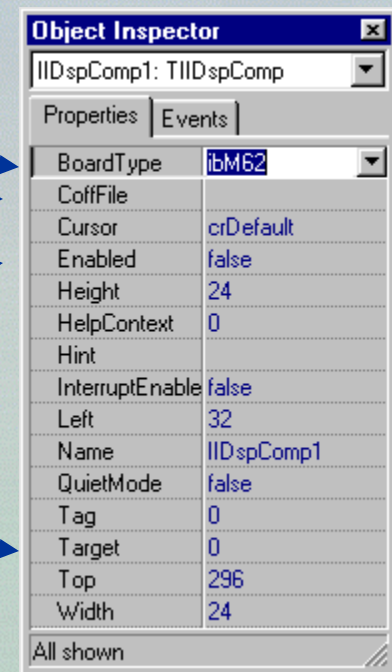
            /* Send 'reps' count back to host */
            write_mailbox(reps, MailboxID);
        }

        /* Could do other processing here while waiting for
         * commands to arrive */
    }
}
```



# Class Project

- Use DspComponent to control target
  - Drop from the Innovative Tab onto the application form
  - View Properties in the Inspector
    - BoardType -- set to ibM62
    - CoffFile -- set to "Demo1.out"
      - File to download
    - Enabled -- set to true
      - Loads DLL on program start
    - Target -- set to 0
      - ID number of M62 in system





# Class Project



- More on DspComponent
  - Hit <Ctrl>-F1 to bring up Help File
  - Public Properties
    - MailboxCount
      - Number of supported mailboxes
    - MailInFull and MailOutEmpty array
      - Flag if data is available from target or has been received by target
    - Mailbox array
      - Read or write data to target. Will wait for data if reading, or wait for last word to be read if writing.



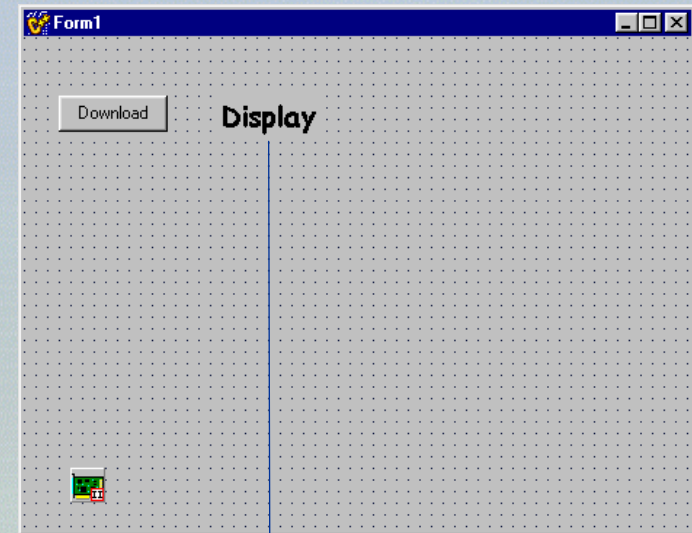
# Class Project

- More on DspComponent
  - Methods
    - BootTarget()
      - Resets and boots target board.
      - Returns true if successful, false on failure
    - Download()
      - Downloads the COFF file. Returns false on failure
      - Returns true if successful, false on failure



# Create the UI

- Add a Button to the Form
  - Set Name to "DownloadBtn"
  - Set Caption to "Download"
- Add a Message Label to the Form
  - Set Name to "Display"
  - We will use this to print messages to the user



You can use the Font property to make the Display label larger. Click on the Font property, then on the '...' button to display a selection dialog.



# The Download Button

- Double-Click on the Download button
  - Skeleton Handler appears in code window
  - Fill in Download Sequence
  - Run the application
- Did "Download Complete!" Appear?

😊 Yes - go on!

😞 No...

- Is the CoffFile property name correct?
- Is the DspComponent Enabled property true?

```
//-----  
void __fastcall TForm1::DownloadBtnClick(TObject *Sender)  
{  
    //  
    // Try to boot the M6x board  
    if (! IIDspComp1->BootTarget())  
    {  
        Display->Caption = "Could not boot target";  
        return;  
    }  
    //  
    // Try to download the target code  
    if (! IIDspComp1->Download())  
    {  
        Display->Caption = "Could not download to target";  
        return;  
    }  
  
    Display->Caption = "Download Complete!";  
}
```

It may not seem like much, but this code will reset the board hardware, find and load the COFF file (also known as the .OUT file), transfer it to the M62 memory and run it.



# Target Synchronization

- The Problem
  - Target may be unready for commands
    - The host may need to wait until the target is configured and ready
  - Coordinate Host and Target
    - Need to know if target has reached some point in the code
- The Solution
  - Use a Read/Write pattern to allow the target to catch up
    - Note: the target uses the opposite order than the host

```
Display->Caption = "Download Complete!";  
//  
// Synchronize with target  
const int MailboxID = 0;  
  
int value = IIDspCompl->Mailbox[MailboxID];  
IIDspCompl->Mailbox[MailboxID] = 0;  
}
```

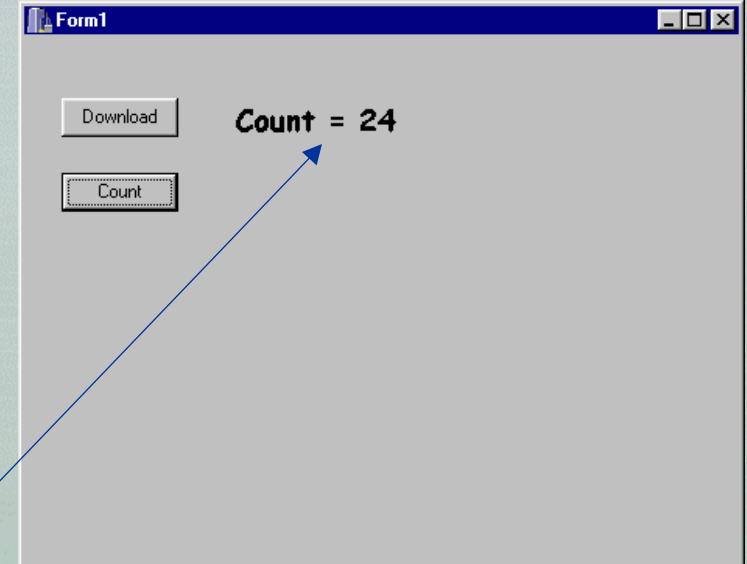
Add this code to the end of the Download button handler you just created.



# Command #1

- The Target program increments a counter each command
  - Drop a new button on the form
  - Rename the button "CountBtn"
  - Change the Caption to "Count"
  - Create a button handler
    - Have it send a '0' command, read the reply, and display it
  - Run the program
    - The display shows a new number each time the button is pressed...

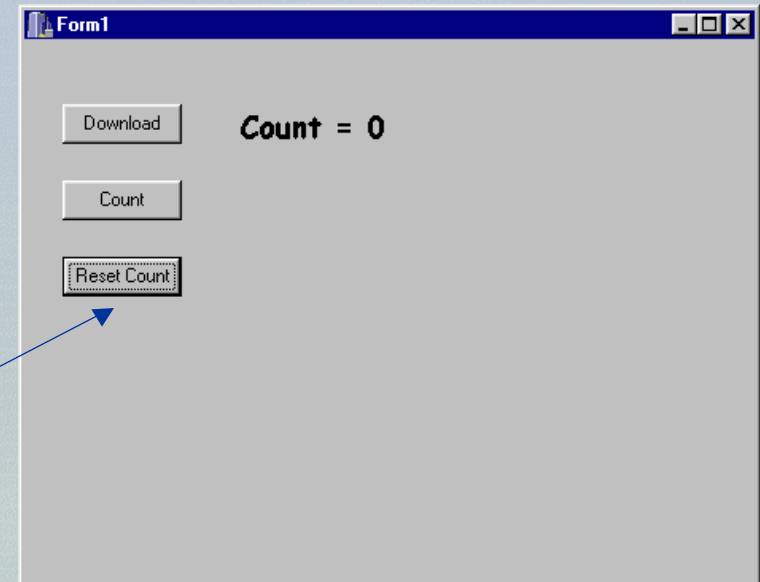
```
//-----  
void __fastcall TForm1::CountBtnClick(TObject *Sender)  
{  
    const int MailboxID = 0;  
    //  
    // Send Command 0  
    IIDspComp1->Mailbox[MailboxID] = 0;  
    // Response...  
    int count = IIDspComp1->Mailbox[MailboxID];  
  
    Display->Caption = "Count = " + AnsiString(count);  
}
```





# Command #2 - Reset Count

- Implement the Reset function
- Create a Reset Button
- Implement its handler
  - Send command code 0xFFFF
  - Fetch the reply
  - Display the count...
- The count resets!



In larger applications, commands can be more complex, but the idea is much the same.